Arbitrarily Accurate Computation with R: The 'Rmpfr' Package

Martin Mächler

maechler@R-project.org (R-Core)
maechler@stat.math.ethz.ch (ETH)

Seminar für Statistik
ETH Zurich Switzerland

ZurichR @ ETH, Jan.19, 2012

Outline

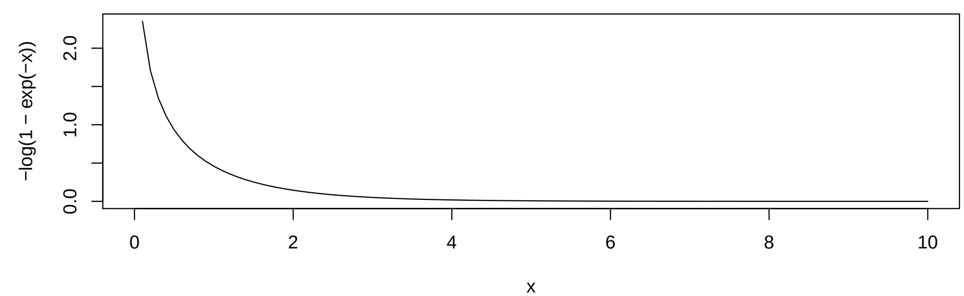
Outline

Logistic regression: Computing "logit()"s, $\log \frac{p}{1-p}$ accurately for very small p, i.e., $p = \exp(-L)$, or

$$\log \frac{p}{1-p} = \log p - \log(1-p) = -L - \log(1 - \exp(-L)),$$

and hence $-\log(1 - \exp(-L))$ is needed, e.g., when p is really really close to 0, say $p = 10^{-1000}$, as then we can only compute $\operatorname{logit}(p)$, if we specify $L := -\log(p) \leftrightarrow p = \exp(-L)$.

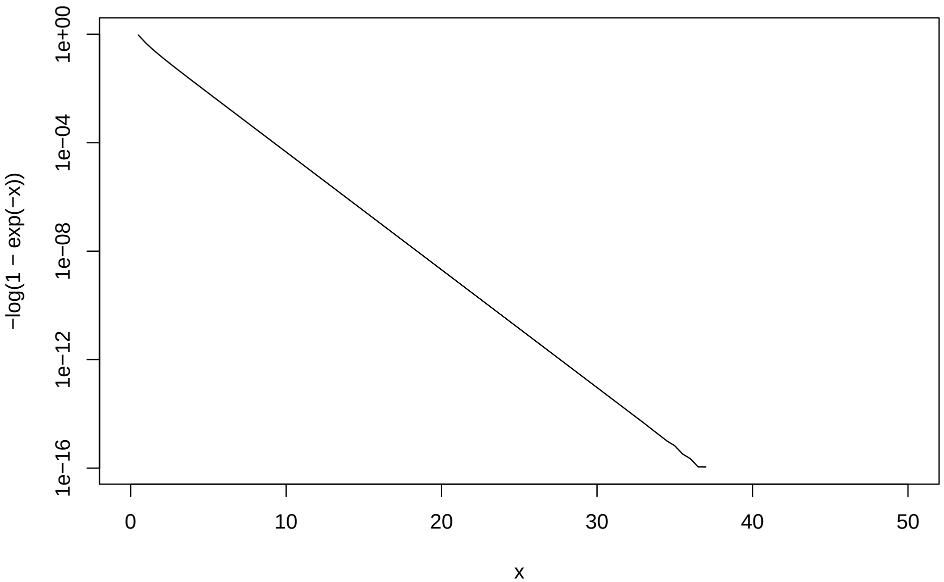
```
> curve(-log(1 - exp(-x)), 0, 10)
```



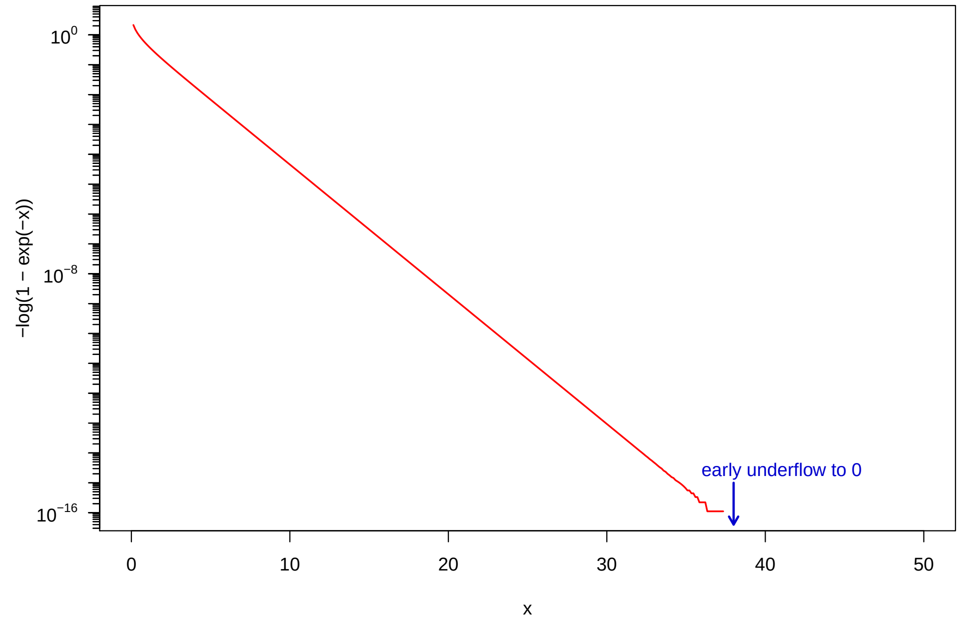
seems fine. —— However, ...

However, further out to 50 (and on a log scale), we observe

```
> curve(-log(1 - exp(-x)),  0, 50, log="y")
```



which shows early underflow.

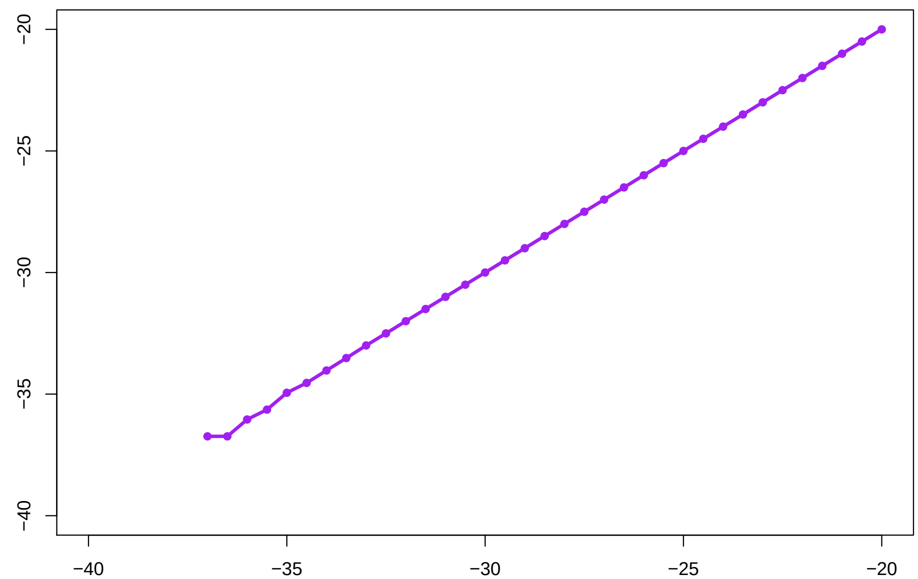


What did happen? Look at

```
> x <- -40:-35
>     -log(1 - exp(x))

[1] 0.000000e+00 0.000000e+00 0.000000e+00 1.110223e-16 2.220446e-16
[6] 6.661338e-16

> log(-log(1 - exp(x)))# --> -Inf values

[1]      -Inf      -Inf      -Inf -36.73680 -36.04365 -34.94504

> ## ok, how about more accuracy
> x. <- mpfr(x, 120)
> log(-log(1 - exp(x.)))# aha... looks perfect now

6 'mpfr' numbers of precision  120   bits
[1] -39.999999999999999997932904877538241734
[2]  -38.99999999999999999423372196756935807
[3]  -37.99999999999999998430451715981029611
[4] -36.999999999999999957331848579613165434
[5] -35.999999999999999884024061830552087239
[6] -34.999999999999999684744214015307532692
```

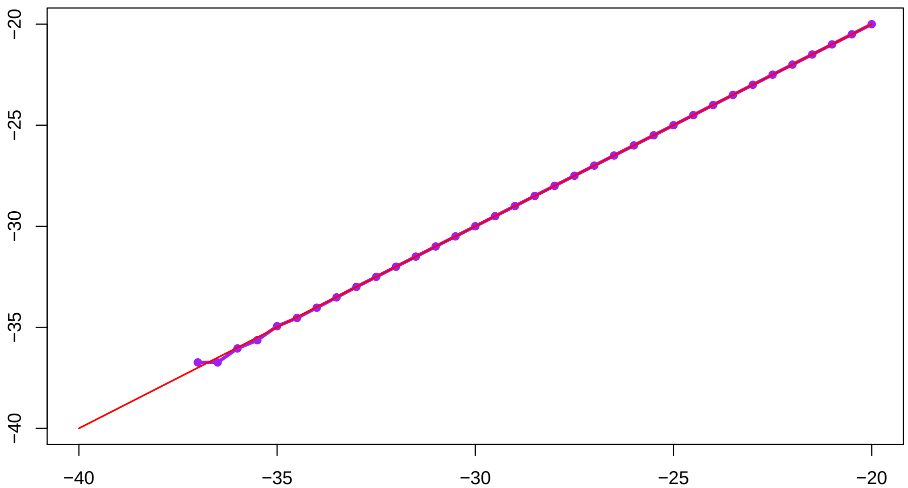

And visually:

```
> x <- seq(-40, -20, by = .5)
> plot(x,x, type="n", ylab="", ann=FALSE)
> lines(x, log(-log(1 - exp(x))), type = "o", col = "purple", lwd=3
```



Now repeat this with “with accuracy”:

```
> x <- seq(-40, -20, by = .5)
> plot(x,x, type="n", ylab="", ann=FALSE)
> lines(x, log(-log(1 - exp(x))), type = "o", col = "purple", lwd=3
> x. <- mpfr(x, 120)
> lines(x, log(-log(1 - exp(x.))), col=2, lwd=1.5)
```



Outline

Exact Factorials and Binomial Coefficients

In combinatorics or when computing series, work with *exact* factorials or binomial coefficients. E.g.,need all factorials $k!$, for $k = 1, 2, \dots, 24$ or a full row of Pascal’s triangle, i.e., want all $\binom{n}{k}$ for $n = 50$.
With R’s double precision, and if you display its full internal precision,

```
> noquote(sprintf("%-30.0f", factorial(24)))

[1] 620448401733239409999872
```

then it is obviously wrong for $24!$, as its last digits are known to be 0.

Easily get full precision results, by replacing “simple” numbers by “mpfr”s:

```
> ns <- mpfr(5:24, 120) ; factorial(ns)
20 'mpfr' numbers of precision  120   bits
 [1]                       120                       720
 [3]                      5040                     40320
 [5]                    362880                   3628800
 [7]                  39916800                 479001600
 .......
 .......
[13]           355687428096000          6402373705728000
[15]        121645100408832000       2432902008176640000
[17]      51090942171709440000    1124000727777607680000
[19]   25852016738884976640000  620448401733239439360000
```


Or for the 70-th Pascal triangle row, $\binom{n}{k}$ for $n = 70$ and $k = 0, \dots, n$,

```
> chooseMpfr.all(n = 70)
70 'mpfr' numbers of precision  67   bits
 [1]                   70                 2415                54740
 [4]               916895             12103014            131115985
 [7]           1198774720           9440350920          65033528560
[10]         396704524216        2163842859360       10638894058520
 .......
 .......
[25]   6455761770304780752 11173433833219812840 18208558839321176480
[28]  27963143931814663880 40498346384007444240 55347740058143507128
[31]  71416438784701299520 87038784768854708790 100226479430802391940
[34] 109069992321755544170 112186277816662845432 109069992321755544170
 .......
 .......
[67]                54740                 2415                   70
[70]                    1
```

Outline

Alternating Binomial Sums

Alternating binomial sums appear in different contexts and are typically challenging, i.e., currently impossible, to evaluate reliably as soon as n is larger than around $50 - 70$.

The alternating binomial sum $sB(f, n) :=$ `sumBinom(n, f, n0=0)` is (up to sign) equal to the n-th forward difference operator $\Delta^n f$,

$$sB(f, n) := \sum_{k=0}^{n} (-1)^k \binom{n}{k} \cdot f(k) = (-1)^n \Delta^n f, \tag{1}$$

where

$$\Delta^n f = \sum_{k=0}^{n} (-1)^{n-k} \binom{n}{k} \cdot f(k) \tag{2}$$

is the n-fold iterated forward difference $\Delta f(x) = f(x+1) - f(x)$ (for $x = 0$).

computing alternating binomial sums in R

An obvious R implementation of $sB(f, n) = \sum_{k=0}^{n} (-1)^k \binom{n}{k} \cdot f(k)$,

```
> sumBinom <- function(n, f, n0=0, ...) {
+   k <- n0:n
+   sum( choose(n, k) * (-1)^k * f(k, ...))
+ }
> ## and the same for a whole *SET* of  n  values:
> sumBin.all.R <- function(n, f, n0=0, ...)
+    sapply(n, sumBinom, f=f, n0=n0, ...)
```

Will see: gets numerical problems, for relatively small n even for well behaved functions $f(\cdot)$.

The `Rmpfr` version is pretty simple, as well:

```
> sumBinomMpfr
function (n, f, n0 = 0, alternating = TRUE, precBits = 256)
{
    stopifnot(0 <= n0, n0 <= n, is.function(f))
    sum(chooseMpfr.all(n, k0 = n0, alternating = alternating) *
        f(mpfr(n0:n, precBits = precBits)))
}
<environment: namespace:Rmpfr>
```

and has a corresponding version for a full set of n:

Compute sumBinomMpfr(n) for a whole set of 'n' values:

```
> sumBin.all <- function(n, f, n0=0, precBits = 256, ...)
+ {
+   N <- length(n)
+   precBits <- rep(precBits, length = N)
+   ll <- lapply(seq_len(N), function(i)
+            sumBinomMpfr(n[i], f, n0=n0, precBits=precBits[i], ...
+   sapply(ll, as, "double")
+ }
```

((Note that sapply(.) is not directly applicable, because its "simplify" part behaves wrongly with vectors of "mpfr" numbers.))

Comparison "double" vs "mpfr":

For comparison, computing the alternating binomial sum,

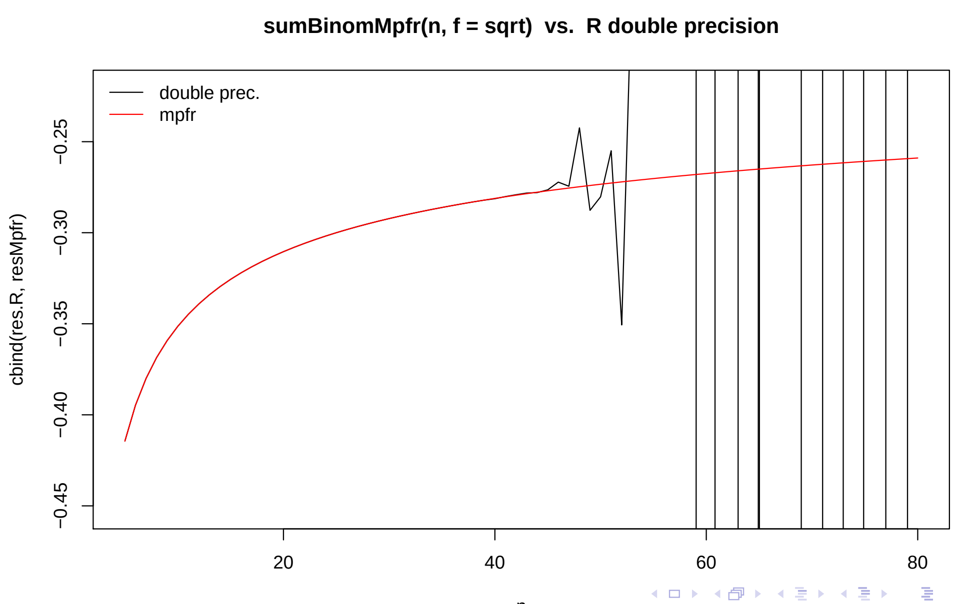
$$sB(f,n) := \sum_{k=0}^{n} (-1)^k \binom{n}{k} \cdot f(k),$$

now try the simple $f(x) = \sqrt{x}$, i.e., in R, sqrt(x):

```
> nn <- 5:80
> system.time(res.R   <- sumBin.all.R(nn, f = sqrt)) ## instant!

   user  system elapsed
  0.002   0.000   0.002

> system.time(resMpfr <- sumBin.all  (nn, f = sqrt)) ## ~2 seconds

   user  system elapsed
  1.525   0.007   1.573
```



```
> matplot(nn, cbind(res.R, resMpfr), type = "l", lty=1,
+         ylim = extendrange(resMpfr, f = 0.25), xlab = "n",
+         main = "sumBinomMpfr(n, f = sqrt)  vs.  R double precisio
> legend("topleft", leg=c("double prec.", "mpfr"), lty=1, col=1:2, l
```



Outline

Capabilities of Rmpfr – a Glimpse

"All" R arithmetic and math functions just work with "mpfr" numbers:
Via "Group" S4 methods

```
> getGroupMembers("Arith")
[1] "+"   "-"   "*"   "^"   "%%"  "%/%" "/"
> getGroupMembers("Compare")
[1] "==" ">"  "<"  "!=" "<=" ">="
> getGroupMembers("Math")
 [1] "abs"      "sign"     "sqrt"     "ceiling"  "floor"    "trunc"
 [7] "cummax"   "cummin"   "cumprod"  "cumsum"   "exp"      "expm1"
[13] "log"      "log10"    "log2"     "log1p"    "cos"      "cosh"
[19] "sin"      "sinh"     "tan"      "tanh"     "acos"     "acosh"
[25] "asin"     "asinh"    "atan"     "atanh"    "gamma"    "lgamma"
[31] "digamma"  "trigamma"
```


Capabilities of Rmpfr — 2 —

In addition to the basic arithmetic (including all "Math" functions!), based on the MPFR C library, Rmpfr provides arbitrarily precise versions of

- Bessel functions $j_n(x)$, $y_n(x)$, and $Ai(x)$
- Error functions `erf(x)`, and `erfc(x)`, or equivalently, `pnorm(x)` and `pnorm(x, lower.tail=FALSE)`.
- Riemann's $\zeta(x)$ =`zeta(x)`,
- Exponential integral `Ei(x)`
- Dilogarithm $\mathrm{Li}_2(x)$ =`Li2(x)`

Capabilities of Rmpfr — 3 —

- Arbitarily precise numerical integration (via Romberg), via our `integrateR()`
- Arbitarily root finding (and hence numerical *inverse* function), via `unirootR()`.

High precision Matrices

Can also do simple arithmetic with "mpfrMatrix" and "mpfrArray" objects, e.g.

```
> head(x <- mpfr(0:7, 64)/7)
6 'mpfr' numbers of precision  64   bits
[1]                        0 0.142857142857142857141 0.28571428571428571428
[4] 0.428571428571428571436 0.571428571428571428564 0.71428571428571428569
> mx <- x ; dim(mx) <- c(4,2)
> mx[ 1:3, ] + c(1,10,100)
'mpfrMatrix' of dim(.) =  (3, 2) of precision  64   bits
     [,1]                   [,2]
[1,] 1.00000000000000000000 1.57142857142857142851
[2,] 10.1428571428571428570 10.7142857142857142860
[3,] 100.285714285714285712 100.857142857142857144
```

We can transpose or multiply such matrices, e.g.,

```
> t(mx) %*% 10^(1:4)

'mpfrMatrix' of dim(.) =  (2, 1) of precision  64   bits
     [,1]
[1,] 4585.71428571428571441
[2,] 10934.2857142857142856
```

or

```
> crossprod(mx)

'mpfrMatrix' of dim(.) =  (2, 2) of precision  64   bits
     [,1]                     [,2]
[1,] 0.285714285714285714282 0.775510204081632653086
[2,] 0.775510204081632653086  2.57142857142857142851
```

and apply works too :

```
> (s7 <- apply(7 * mx, 2, sum))

2 'mpfr' numbers of precision  64   bits
[1]  6 22
```

and, note that `all.equal()` methods are provided, as well:

```
> all.equal(s7, c(6,22), tol = 1e-40) # note the tolerance!

[1] TRUE
```


We can transpose or multiply such matrices, e.g.,

```
> t(mx) %*% 10^(1:4)

'mpfrMatrix' of dim(.) =  (2, 1) of precision  64   bits
     [,1]
[1,] 4585.71428571428571441
[2,] 10934.2857142857142856
```

or

```
> crossprod(mx)

'mpfrMatrix' of dim(.) =  (2, 2) of precision  64   bits
     [,1]                     [,2]
[1,] 0.285714285714285714282 0.775510204081632653086
[2,] 0.775510204081632653086  2.57142857142857142851
```

and apply works too :

```
> (s7 <- apply(7 * mx, 2, sum))

2 'mpfr' numbers of precision  64   bits
[1]  6 22
```

and, note that `all.equal()` methods are provided, as well:

```
> all.equal(s7, c(6,22), tol = 1e-40) # note the tolerance!

[1] TRUE
```


Outline


```
> toLatex(sessionInfo())
```

- R version 2.14.1 Patched (2012-01-17 r58138), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=de_CH.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=de_CH.UTF-8`, `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=de_CH.UTF-8`, `LC_PAPER=C`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=de_CH.UTF-8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: Rmpfr 0.4-5, sfsmisc 1.0-19
- Loaded via a namespace (and not attached): gmp 0.5-0, tools 2.14.1

```
> packageDescription("Rmpfr")

Package: Rmpfr
Type: Package
Title: R MPFR - Multiple Precision Floating-Point Reliable
Version: 0.4-5
Date: 2012-01-12
Author: Martin Maechler
Maintainer: Martin Maechler <maechler@stat.math.ethz.ch>
Depends: methods, R (>= 2.11.0)
SystemRequirements: gmp (>= 4.2.3), mpfr (>= 3.0.0)
SystemReqsNotes: MPFR (MP Floating-Point Reliable Library,
       http://mpfr.org/) and GMP (GNU Multiple Precision library,
       http://gmplib.org/), see README
Imports: gmp
Suggests: gmp, polynom
SuggestNotes: 'polynom' is only needed for vignette
URL: http://rmpfr.r-forge.r-project.org/
Description: Rmpfr provides S4 classes and methods for arithmetic
       including transcendental ("special") functions for
       arbitrary precision floating point numbers. To this end, it
       interfaces to the LGPL'ed MPFR (Multiple Precision
       Floating-Point Reliable) Library which itself is based on
       the GMP (GNU Multiple Precision) Library.
License: GPL (>= 2)
```


Outline

Conclusion

- The package `Rmpfr` allows to use arbitrarily high precision numbers instead of R 's double precision numbers in many R computations and functions.
- This is achieved by defining S4 classes of such numbers and vectors, matrices, and arrays thereof, where all arithmetic and mathematical functions work via the (GNU) MPFR C library, where MPFR is acronym for "**M***ultiple* **P***recision* **F***loating-Point* **R***eliably*". MPFR is Free Software, available under the LGPL license, and itself is built on the free GNU Multiple Precision arithmetic library (GMP).
- Consequently, by using `Rmpfr`, you can often call your R function or numerical code with mpfr–numbers instead of simple numbers, and all results will automatically be much more accurate.

Executive Summary

- Double precision accuracy (almost 16 digits) is not always sufficient
- `Rmpfr` is here for arbitrarily precision computations in R .
- Many R functions — **when** `source()`**d** — will work with "mpfr"-numbers automagically

That's all folks — with thanks for your attention!

Martin Mächler – maechler@R-project.org

Executive Summary

- Double precision accuracy (almost 16 digits) is not always sufficient
- `Rmpfr` is here for arbitrarily precision computations in R .
- Many R functions — **when** `source()`**d** — will work with "mpfr"-numbers automagically

That's all folks — with thanks for your attention!

Martin Mächler – maechler@R-project.org